APPLICATION FOR A UNITED STATES PATENT

For:

VARIOUS METHODS AND APPARATUSES FOR INTERFACING OF A PROTOCOL MONITOR TO PROTOCOL CHECKERS AND FUNCTIONAL CHECKERS

Inventors:
Terrence Staton, Herve Alexanian, and
Jeffrey Ebert

Prepared by:

BLAKELY SOKOLOFF TAYLOR & ZAFMAN LLP
32400 Wilshire Boulevard
Los Angeles, CA 90025-1026
(408) 720-8300

Attorney's Docket No.: 002998.P033

"Express Mail" mailing label number: EV336584735US
Date of Deposit: July 24, 2003
I hereby certify that I am causing this paper or fee to be deposited with the United States
Postal Service "Express Mail Post Office to Addressee" service on the date indicated above and that this paper or fee has been addressed to the Box Provisional Application, Commissioner for Patents, Washington, D. C. 20231
Deborah A. McGovern
(Typed or printed name of person mailing paper or fee)
D McGovern
(Signature of person mailing paper or fee)
July 24, 2003

VARIOUS METHODS AND APPARATUSES FOR INTERFACING OF A PROTOCOL MONITOR TO PROTOCOL CHECKERS AND FUNCTIONAL CHECKERS

NOTICE OF COPYRIGHT

BACKGROUND OF THE INVENTION

[002] The Intellectual Property (IP) cores may be circuitry, embedded memories, communication links, a processor, etc., having self-contained designed functionality. Additionally, IP cores may be implemented in a variety of ways, and may be fabricated on a single integrated circuit such as a System on a Chip.

[003] Various interconnect protocols exist such as Open Core Protocol™ (OCP), Peripheral Component Interconnect, Media Independent Interface, and Advanced High-performance Bus (AHB). Some interconnect protocols define a bus-independent interface for IP cores. System-on-a-Chip IP core interfaces can be described in this consistent format, such as a simple peripheral core, a high-performance microprocessor, or even an on-chip communication subsystem. Unique for each core, the IP core interface includes all of the signals required to describe IP core communication including data flow, control, and verification and test signals. An IP core interface assists system verification and testing by providing a firm boundary around each IP core that can be observed, controlled, and validated.

[004] System on Chip (SoC) designs use bus protocols for high performance data transfer among the Intellectual Property cores located on that chip. These interconnect protocols may incorporate advanced features such as pipelining, burst and split transfers, etc. Many of the modules connected to the bus are vendor provided IP cores in which the designs for these IP cores may not be made available. The SoC bus protocol between these IP cores as well as the functionality provided by each IP core needs to be debugged and have various design values verified.

[005] An approach to verifying digital hardware during a simulation run may be to utilize software monitors and software checkers written in C++. The monitor serves the purpose of collecting data from a hardware interconnect and making it available to a checker. An interconnect can be a set of wires that carries data and control information between two IP cores. The checker may model the functionality of an IP core using the data from the monitors as input. The checker communicates to the monitor that it needs data and establishes a single location to send that data. These monitors collect data from a hardware interconnect that is attached to the IP core. The checker compares the expected results of the IP core it is modeling to the actual data by using the data collected through the same monitors. Typically, a monitor exists for each of the interfaces of an IP core. The data collected from each of these monitors is passed to the checker for that specific IP core. A monitor may also be located inside or external to an IP core.

[006] This typical approach quickly becomes problematic in hardware designs where several IP cores are connected via hardware interconnects. A monitor on each interconnect could potentially have to provide various items of data to several checkers. For instance, a monitor might need to provide data to a master core checker, a slave core checker and a protocol checker all at the same time. The typical approach is that a monitor does not share data required by several checkers. Each checker has its own monitor to retrieve the needed data.

[007] Further, the typical approach may have protocol monitors write protocol data to a file during a simulation run. After a simulation run completes, then a series of tools run to analyze the data contained in the trace files to detect errors and display performance data. This process lacks the ability to check for errors in a design during a simulation run. It may be desirable to terminate a simulation a few clock cycles after an error occurs in a hardware design in order to decreases costs both in terms of required CPU time and time between successive simulation runs.

SUMMARY OF THE INVENTION

[008] Various methods and apparatuses are described in which a software programming interface connects one or more functional checker components and one or more protocol checker components to an interconnect monitor component. A computer readable medium stores code for the one or more functional checker components for Intellectual Property cores, one or more protocol checker components, the interconnect monitor component, and the software programming interface. The monitor component has code to build data structures containing protocol data types requested by a checker component and code on where to deliver data based upon a particular type of data requested by the checker component.

BRIEF DESCRIPTION OF THE DRAWINGS

[009] The drawings refer to embodiments of the invention in which:

figure 1 illustrates a block diagram of an embodiment of one or more interconnect monitor components, one or more functional checkers components for Intellectual Property (IP) cores, and one or more protocol checker components using a programming interface to cooperate to verify and debug an IP core design;

figure 2 illustrates a block diagram of an embodiment of one or more interconnect monitor components, one or more functional checkers components for Intellectual Property (IP) cores, and one or more protocol checker components using a well-defined programming interface to allow an error to be detected during a simulation run; and

figure 3 illustrates a flow diagram of an embodiment of the programming interface connecting one or more protocol checker components and one or more functional checker component for IP cores to a monitor component to debug and verify an IP core design.

[0010] While the invention is subject to various modifications and alternative forms, specific embodiments thereof have been shown by way of example in the drawings and will herein be described in detail. The invention should be understood to not be limited to the particular forms disclosed, but on the contrary, the intention is to cover all modifications, equivalents, and alternatives falling within the spirit and scope of the invention.

DETAILED DISCUSSION

[0011] In the following description, numerous specific details are set forth, such as examples of specific data signals, named components, connections, computer programming languages, etc., in order to provide a thorough understanding of the present invention. It will be apparent, however, to one skilled in the art that the present invention may be practiced without these specific details. In other instances, well known components or methods have not been described in detail but rather in a block diagram in order to avoid unnecessarily obscuring the present invention. Further, specific numeric references such as first protocol checker component, may be made. However, the specific numeric reference should not be interpreted as a literal sequential order but rather interpreted that the first protocol checker component is different than a second protocol checker component. Thus, the specific details set forth are merely exemplary. The specific details may be varied from and still be contemplated to be within the spirit and scope of the present invention.

[0012] In general, various methods and apparatus are described in which a software programming interface connects one or more functional checker components and one or more protocol checker components to an interconnect monitor component. A computer readable medium stores code for the one or more functional checker components for Intellectual Property (IP) cores, one or more protocol checker components, the interconnect monitor component, and the software programming interface. The monitor component has code to build data structures containing protocol data types requested by a checker component and code on where to deliver data based upon a particular type of data requested by the checker component. The monitor component has code to create protocol data objects that tracks within the protocol data object whether the data is still being used by another checker component and when the protocol data object should be deleted. The monitor component calls methods defined in the code of the checker component as data becomes available during the simulation run to allow an error to be detected as soon as possible during the simulation run.

[0013] Figure 1 illustrates a block diagram of an embodiment of one or more interconnect monitor components, one or more functional checker components for Intellectual Property (IP) cores, and one or more protocol checker components using a programming interface to cooperate to verify and debug an IP core design or a collection of such cores in a system. Several IP cores, such as RAM 102, Logic 104, etc. on a System on a Chip 100 may be connected via various hardware interconnects 106. A hardware interconnect such as a first hardware interconnect 108 may be a set of wires that connects two or more hardware IP cores together. A software monitor component, such as the first monitor component 110, is programmed to collect data from a hardware interconnect that is attached to an IP core, such as the Digital Signal Processor 112. The software monitor component makes the monitored data available to a checker component. A functional checker component for an IP core, such as the first IP core checker 112, models the functionality of an IP core using the data from the monitor component as input. A protocol checker generation component determines the type of protocol interconnect that is being used. The protocol checker generation software component may be used to loop through all the hardware interconnects in the design and to create an instance of an appropriate protocol checker for that connection, such as the first protocol checker 114. For functional checkers, a product specific code separate from the monitors and protocol checkers may loops through all the cores and spawn instances of functional checkers for them. The actual checks between the protocol checker component and the monitor component may be determined by the protocol itself. For example, a data handshake-phase starting on a thread that has no previous corresponding request phase on the same thread is an unrecoverable error for an OCP software monitor. Two or more checker components may be used when verifying the same configurable interconnect protocol, such as OCP, through the same monitor component. The monitor component supplies data to these two or more checker components. For example, the first monitor 110 supplies data to the first functional checker component for an IP core 112, the first protocol checker component, and a second protocol checker component 116. A well-defined programming interface exists for the multiple software checker components per software monitor component. A checker component, such as a protocol checker, functional checker, system checker, or similar checker uses the modeled protocol data for input stimulus and output verification. The functional checker component detects errors in an IP core design and communicates those errors to a user during a simulation run.

[0014] The software programming interface connects one or more functional checker components for IP cores, such as the first functional checker component 112 and a second functional checker component 118, and one or more protocol checker components, such as a first protocol checker component 114 and a second protocol checker component 116, to a single interconnect monitor component such as the first monitor component 110. The software programming interface enables a checker component to request callback services for specific items of data provided by the interconnect monitor. The interconnect monitor invokes a routine by calling methods defined in the code of the checker components as data becomes available during the simulation. Each item of data passed to the checker components may be self-maintaining and deletes itself once all of the checker components are no longer referencing that item of data. The software programming interface enables a clear separation of code between the interconnect monitors and checker components receiving data from that interconnect monitor.

[0015] One or more computer readable mediums, such as memories, may store the code for the one or more interconnect monitor components, the one or more functional checker components for IP cores, the one or more protocol checker components, and the software programming interface. A checker component may be modeled to represent the functional checker of an IP core, a protocol checker, a higher level construct modeled to debug an IP core design, or other type of checker. The software interface between the monitor component and checker component does not impose any restrictions on the type of checker component that can be constructed. Implementing a checker component to model functionality of an IP core for debug purposes may use the software interface to a monitor in the same manner as a protocol checker or any other type of checker. The one or more interconnect monitor components, one or more functional checker components for IP core components, and one or more protocol checker components cooperate to verify and debug an IP core design and its interactions with other IP cores.

[0016] The software programming interface, protocol checker components, functional checker components, and interconnect monitor component may be coded in an object-oriented programming language, such as Java, C++, Smalltalk, object-oriented versions of Pascal, or other similar language. Note, another programming language may be used to create the software programming interface, protocol checker components, functional checker components, and interconnect monitor component, although the features found in object-oriented languages will be used for describing an embodiment of this invention.

[0017] The interconnect monitor component has code to build object oriented data structures containing protocol data types requested by the checker component. For example, the first interconnect monitor component 110 has code to build object oriented data structures containing protocol data types requested by the first protocol checker component 114 and the first functional checker component for an IP core 112. The interconnect monitor component has code that determines where to deliver the collected data based upon the particular type of data requested by the checker component. The software interface component defines both the data type of a data structure and the types of operations, such as functions, that can be applied to the data structure. In this way, the data structure becomes an object that includes both data and functions that are specific to the protocol being modeled. The software interface includes a set of base classes for the data structures. The monitor component builds data structures from the data types defined in the software interface component for all types of objects, checkers, etc., that have registered with the monitor and that need protocol data from the monitor interconnect component. The monitor component may create data structures placed into shared pointers, which are then passed from the monitor component to pass data from a single monitor component to one or more checker components. One shared pointer could be passed to several objects such as checker components. The monitor component may have code to create protocol data objects that are instances of the data structure from the software interface component. The protocol data object tracks within itself whether the data is still being used by another checker component and when the protocol data object should be deleted via a reference counted pointer. Thus, the data is self-maintaining because the data object deletes itself when the last checker component referencing the data object no longer needs that data.

[0018] The software interface component also includes a set of base classes to assist the monitor component in implementing callback services. The set of base classes of the programming interface allows implementation across two or more types of interconnect protocols. The set of base classes of the software interface can be used by multiple monitor component implementations, one for each interconnect protocol to be supported.

[0019] The monitor component may contain a template abstraction sensitive to the particular data type being requested by the checker component when that checker component registers a data callback service with the monitor component. The particular data types supported by a monitor component may be based on the protocol of the hardware interconnect being monitored. The monitor component calls method(s) defined in the code of the checker component(s) as data becomes available during the simulation run to allow an error to be detected during a simulation run.

[0020] When first initialized, a protocol checker component retrieves the instance of the monitor that will create data structures for the protocol of the hardware interconnect the checker has been assigned. A checker component, such as the first protocol checker 114, directs the monitor component at the start of a simulation run on the type of data requested and the one or more locations to send that data in order to verify the hardware interconnect. The checker directs the monitor on the type of data requested and the monitor reports an exception if that type of data is not available for that particular protocol interconnect. A checker component may also register and unregister callback services at anytime during a simulation run. Utilities other than checkers such as performance analyzers might also use this feature.

[0021] The monitor contains code to obtain protocol data in an object-oriented form and then relays that protocol data to several checker components or even several locations within a single checker component. The monitor relays that information using an object-oriented call, such as calling a dereferenced pointer to a method, to send the data to the checker component specified locations. The monitor may use the pointer to the method sent from the checker component to identify the data type that was requested as well as the location to which the data with that type should be delivered. For example, two instances of the protocol data may be created to send a particular data type to two different locations in the same checker component. Each checker instance sends that particular data type to a separate location. Also, one data item may go to two or more locations and each location is within a different checker component.

[0022] The monitor component supplies callback services that are sensitive to the particular data type requested by a checker component. The checker component provides unique delivery locations for each requested data type to the monitor component. The checker component provides the particular data types requested from the monitor to the monitor component at the time of registration for that data. The software interface cooperates with code in a hardware interconnect monitor to enable a checker component to request callback services for specific items of data provided by one or more hardware interconnect monitors.

[0023] Once the data is accessible, the monitor knows where to deliver the data based on the type of data. Note, for more complex data types it may take many cycles of data collection to make one data object available. The types of data depend on the particular interconnect protocol supported by the monitor. The data types may be defined in the software interface. Data types vary within a single protocol as well as data types vary in different protocols. One software interface may be defined for each interconnect protocol. For example, OCP interconnect protocol types of data may be request phase data, burst transaction data, etc. Advanced High-performance Bus (AHB) interconnect protocol, types of data may be transfer data, burst transaction data, etc. The monitor component communicates the value of the data as the data becomes available. The monitor also communicates the time when the data became available such as in synchronization clock cycles, edge sensitive data events both positive and negative, simulation time typically in nanosecond, or some other similar indication of when the data became available.

[0024] Figure 2 illustrates a block diagram of an embodiment of one or more interconnect monitor components, one or more functional checkers components for Intellectual Property (IP) cores, and one or more protocol checker components using a well-defined programming interface to allow an error to be detected during a simulation run. Two monitor components, a first monitor component 210 and a second monitor component 220, provide protocol data to three checker components, 214, 218, 222. The first monitor component 210 provides data to the first protocol checker component 214 and the first functional checker component 218. The second monitor component 220 provides data to the second protocol checker component 222 and the first functional checker component 218.

[0025] The programming interface allows code separation between checkers and monitors to remain separate and create the required flexibility in solving an undetermined problem space. The programming interface may be a language and message format used by the application program to communicate with the communications protocol. The programming interface may be implemented by writing function calls in the program, which provide the linkage to the required subroutine for execution such as callback services and data passing structures. The combination of data passed in well-defined structure and data callback services enables clear and distinct separation between the checker code and the monitor code. The programming interface between the monitors and checker components allows significant gains in verification efficiency.

[0026] The data passing structure and call back services may be implemented in C++.

[0027] The data passed from the monitor components to the several checker components may have a well-defined structure. Monitors collect protocol data into data structures that use shared-pointers for use by checkers. The term data structure may refer to a scheme for organizing related pieces of information. The basic types of data structures may include: files, lists, arrays, records, trees, tables, etc. In particular, C++ shared-pointers may be particularly useful for passing the data from a single monitor component to several checker components. By using shared-pointers, neither the monitor component nor checker components needs to track data usage by all checkers that receive the data.

[0028] A monitor component may only generate data structures as needed to supply data to the checker components. The sum total of requests for data by the checker components to that monitor component is not a maximal set of data types that can be provided by the first monitor component. Thus, the monitor component may be configured to produce only the data structures requested by the checker components at registration, rather than producing every potential data structure that has a possibility of being requested by a checker component in a simulation run. This allows the monitor to perform more efficiently when a small subset of data types is requested by the checker components.

[0029] Checker components selectively register callback services to the monitor components for the protocol data that the checker component requests in performing its checks. The checker component may request data callback services of a monitor component via a C++ template abstraction that is sensitive to the data type being requested by that checker component.

[0030] The data structures used by the programming interface can be defined for each hardware interconnect protocol that is supported. Either a maximal or minimal subset of a protocol can be used when defining the data structures. The use of a maximal or minimal subset depends upon the requirements of what protocol data is to be passed between the checkers and a monitor. For most interconnect protocols, a minimal subset may consist of a signal.

[0031] An example of a signal data structure created by an example monitor component can be summarized with the following example protocol.

```
class Cmd : public SignalT {
    const double & getTime() { return m_dtime; }
    const int & getCycle() { return m_ncycle; }
  private:
    double m_dtime;
    int m_ncycle;
};
```

[0032] The data structure contains the time at which the Cmd data was captured. This may be particularly useful to checker components as it can work with "snap-shots" of data instead of continually having to store state.

[0033] In the example protocol, a higher-level construct of data structures may be defined using the lower level data constructs. For example, a command group may be summarized with the following:

```
class CmdGroup {
  public:
    const double & getTime();
    const int & getCycle();

    SharedPointerT<Cmd>  m_hCmd;
    SharedPointerT<Data> m_hData;
    SharedPointerT<Accept>  m_hAccept;
};
```

[0034] The higher-level constructs may use shared-pointers. The use of shared-pointers eliminates the complex need of determining exactly when data is no longer needed either directly by a checker or through a higher-level data construct still in use by a checker. For example, if a checker has a CmdGroup data object then Cmd, Data and Accept data object referenced by that CmdGroup must be valid while the CmdGroup is in use. Alternatively, other ways are possible to eliminate no longer useful code such as storing data usage into a linked list or array with an algorithm that will scan the data usage structure for elements that are no longer need.

[0035] In the example protocol, an even higher-level construct may be summarized with the following.

```
class Request {
 public:
  const double & getStartTime() { return m_dstartTime; }
  const double & getEndTime() { return m_dendTime; }
  const int & getStartCycle() { return m_nstartCycle; }
  const int & getEndCycle() { return m_nendCycle; }
  const bool & isComplete() { return m_bisComplete; }

  SharedPointerT<CmdGroup> m_hcmdGroup;
 private:
  double m_dstartTime;
  double m_dendTime;
  int m_nstartCycle;
  int m_nendCycle;
  bool m_bisComplete;
  };
```

[0036] The Request class allows a command to take more than one cycle to process. Thus, a check exists to detect when the command has completed. The Request class can then be used in even higher-level constructs of this example protocol. The members of each class and procedures executed when an object receives a message, such as a method, of each class are tailored to the needs of a checker component. Checker implementation time may be reduced by making access to the received data easier. The data structures created for the interface may have their usage optimized to the needs of the checker components.

[0037] An example process by which a checker requests data callback services of a monitor may use a C++ template abstraction, which is sensitive to the data type being requested by the checker. Through the process of abstraction, a verification engineer hides all but the relevant data about an object in order to reduce complexity and increase efficiency. With an abstraction, the object that remains is a representation of the original, with unwanted detail omitted. The resulting object itself can be referred to as an abstraction, meaning a named entity made up of selected attributes and behavior specific to a particular usage of the originating entity.

[0038] The checker component supplies several methods, one per data type to be requested, to a monitor component that will later be called back when specific data for that method is created by the monitor. For example, a checker component might have the following method that uses a data type from the above example protocol.

```
void Checker::requestCallback(SharedPointerT< Request > hrequest) {
    }
```

[0039] A pointer to this method along with a reference to the class instance can be sent to a monitor component via a registration method. A pointer to this particular method have the following type in C++.

```
void (Checker::*)( SharedPointerT< Request >)
```

[0040] The monitor component passes a value, such as an argument, between programs, subroutines or functions. The monitor can use the argument type of the pointer to the method to handle data abstraction for the callback services. A template method to register a callback with a monitor may be summarized with the following prototype.

```
template<class checkerObjT, typename callbackArgT>
void registerCallback(
checkerObjT & checkerObj,
void (checkerObjT::*pcallbackMethod)( callbackArgT);
```

[0041] Through callbackArgT in the above method the monitor can then use a variety of data abstraction approaches to handle the collection and processing of these callbacks to the checkers. Data abstractions of these callback methods include the following four examples:

[0042] 1) Create a STL map container in the monitor keyed to typeid(callbackArgT) with the contents of a defined class which contains both a pointer to a checkerObjT and pointer to a method of type void (checkerObjT::*)(callbackArgT). The monitor completely contains the class objects that hold the pointer to the method and the pointer to the class containing the method. Methods registered with a callbackArgT that are not supported by the monitor may be detected at runtime.

[0043] 2) Create a class that hides type information through the use of a static method that takes two arguments: a void pointer and a callbackArgT. The class may contain both a pointer to the checker object and a pointer to the callback method. The void pointer inside the static method is static_cast to the class's type. This pointer is then used to call the callback method using the .* operator on the checker pointer and the method pointer data members. Methods registered with a callbackArgT that are not supported by the monitor may be detected at compile time.

[0044] 3) Use an extra argument in the callback registration to identify the data type. An enumeration can be constructed to represent each data type. The monitor can key callbacks to checkers using the enumerated data.

[0045] 4) Use a common base class across all protocol data objects. This inherited base class could be used to hide the data types of the derived protocol data objects when processing the callbacks.

[0046] 5) It is possible to mix some of the above four approaches together to achieve a slight variation. For example, approaches 1 and 3 or 1 and 4 can be combined to avoid the usage of the typeinfo library. These approaches are merely examples and many more may be implemented.

[0047] Allowing the checkers to remain separate from the monitor creates the required flexibility in solving an undetermined problem space. When several designs are all interconnected to each other using a varying range of protocols in a user specified configuration then flexibility is of prime importance. The separation allows maximal code reuse across any conceivable problem space. Common base code for the monitor component and the checker components may be written to facilitate reuse of the code across two or more types of interconnect protocols each utilizing different protocols.

[0048] A greater portion of the verification software, the monitor, and protocol interface code, is reused across several design projects. When a new previously unimplemented protocol is encountered an existing set of base classes for the data structures and monitor callback services enables quick development of monitors and checkers.

[0049] Monitor components may take the responsibility of collecting high-level protocol data objects so that verification engineers do not have to write similar code to handle high level protocol constructs across different checker components. The verification engineer writing checkers may focus on modeling rather than the protocols of the hardware interconnects. Verification engineers may debug one protocol monitor for each interconnect protocol to lessen the debug time of the verification software.

[0050] In an embodiment, a machine-readable medium may have stored thereon information representing the apparatuses and/or methods described herein. A machine-readable medium includes any mechanism that provides (e.g., stores and/or transmits) information in a form readable by a machine (e.g., a computer). For example, a machine-readable medium includes read only memory (ROM); random access memory (RAM); magnetic disk storage media; optical storage media; flash memory devices; DVD's, electrical, optical, acoustical or other form of propagated signals (e.g., carrier waves, infrared signals, digital signals, EPROMs, EEPROMs, FLASH, magnetic or optical cards, or any type of media suitable for storing electronic instructions. Slower mediums could be cached to a faster, more practical, medium. The information representing the apparatuses and/or methods stored on the machine-readable medium may be used in the process of creating the apparatuses and/or methods described herein. For example, the information representing the apparatuses and/or methods may be contained in an Instance, soft instructions in an IP generator, or similar machine-readable medium storing this information.

[0051] Some portions of the description may be presented in terms of algorithms and symbolic representations of operations on, for example, data bits within a computer memory. These algorithmic descriptions and representations are the means used by those of ordinary skill in the data processing arts to most effectively convey the substance of their work to others of ordinary skill in the art. An algorithm is here, and generally, conceived to be a self-consistent sequence of acts leading to a desired result. The acts are those requiring physical manipulations of physical quantities. Usually, though not necessarily, these quantities take the form of electrical, magnetic, or optical signals capable of being stored, transferred, combined, compared, and otherwise manipulated. It has proven convenient at times, principally for reasons of common usage, to refer to these signals as bits, values, elements, symbols, characters, terms, numbers, or the like.

[0052] It should be borne in mind, however, that all of these and similar terms are to be associated with the appropriate physical quantities and are merely convenient labels applied to these quantities. Unless specifically stated otherwise as apparent from the above discussions, it is appreciated that throughout the description, discussions utilizing terms such as "processing" or "computing" or "calculating" or "determining" or "displaying" or the like, refer to the action and processes of a computer system, or similar electronic computing device, that manipulates and transforms data represented as physical (electronic) quantities within the computer system's registers and memories into other data similarly represented as physical quantities within the computer system memories or registers, or other such information storage, transmission or display devices.

[0053] Figure 3 illustrates a flow diagram of an embodiment of the programming interface connecting one or more protocol checker components and one or more functional checker component for IP cores to a monitor component to debug and verify an IP core design.

[0054] In block 302, a monitor component monitors a hardware interconnect between two or more IP cores to collect protocol data.

[0055] In block 304, the monitor component generates one or more object oriented data structures to pass the protocol data to two or more locations associated with checker components. The monitor component may implement data structures that use shared pointers to pass data to the locations in two or more checker components or two or more locations within a single checker component. For example, two instances of the checker component may be created to send a particular data type to two different locations in the single checker component. Each checker instance sends that particular data type to a separate location. Also, one data item may go to two or more locations and each location is within a different checker component.

[0056] In block 306, the monitor component receives the type of data requested and locations to send that data from a checker component at the start of a simulation run. Checker components may also register and unregister during the simulation run as well.

[0057] In block 308, the monitor component generates a data item that is self maintaining and will delete itself once all checker components are no longer referencing that data item.

[0058] In block 310, the monitor component calls methods defined in the code of the checker components as data becomes available during the simulation run. Thus, the monitor component is programmed to collect data from hardware interconnect and to make the data available to one or more checker components which detect errors in an IP core design and communicate those errors to a user during a simulation run.

[0059] While some specific embodiments of the invention have been shown the invention is not to be limited to these embodiments. For example, one skilled in the art will recognize that all types of objects can use the protocol data from the monitor component and request data from the monitor component for other kinds of applications rather than just checking. For example, a disassembler can be constructed that needs the some of the protocol data. Two or more functional checker components may interface the same monitor with or without a protocol checker component interfacing that monitor. The invention is to be understood as not limited by the specific embodiments described herein, but only by the scope of the claims.